Advanced Programming Practices

SOEN 6441

Professor: Dr. Khaled Jababo

TEAM #: GROUP_U_B

NAME	STUDENT ID
Arjuman Mansuri	40076180
Goutham Gopal Raje	40054007
NiralKumar Lad	40080612
Mudra Parikh	40068098
Sai Sharan Chitta	40056518
Suraj Reddy Yellu	40057753

Table of Contents

1. Introduction

RISK Game is a desktop based application game developed in java. The objective of this game is to attack and capture all the territories on a given map in order to win the game and become the ruler of the continent. The game has various different maps to play on and can be played by 2 to 6 players. The rules of the game are slightly different depending on the number of players. Risk game consists of a connected graph map representing various maps, where each node is a country and each edge represents adjacency between countries.

2. Architectural Design

This game is designed using the Model-View-Controller (MVC) design pattern. This pattern helps to divide each aspect of the game into either of three roles which helps to determine the functionality of each object in the application and how they would communicate with the rest of the objects. Each objects are present in separate classes and they have abstract boundaries and communicates with other classes across the boundaries.

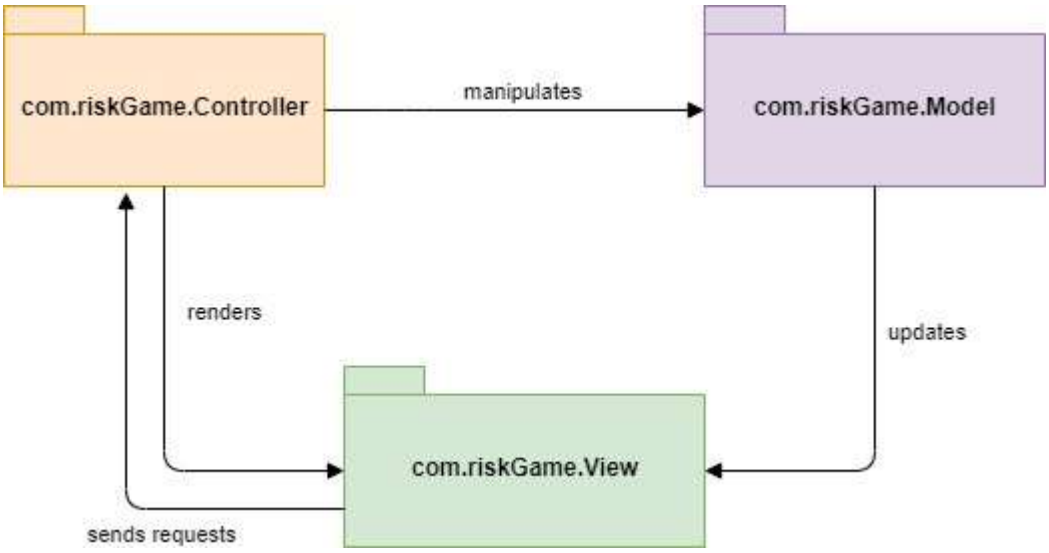


Fig. (1) Basic MVC Model

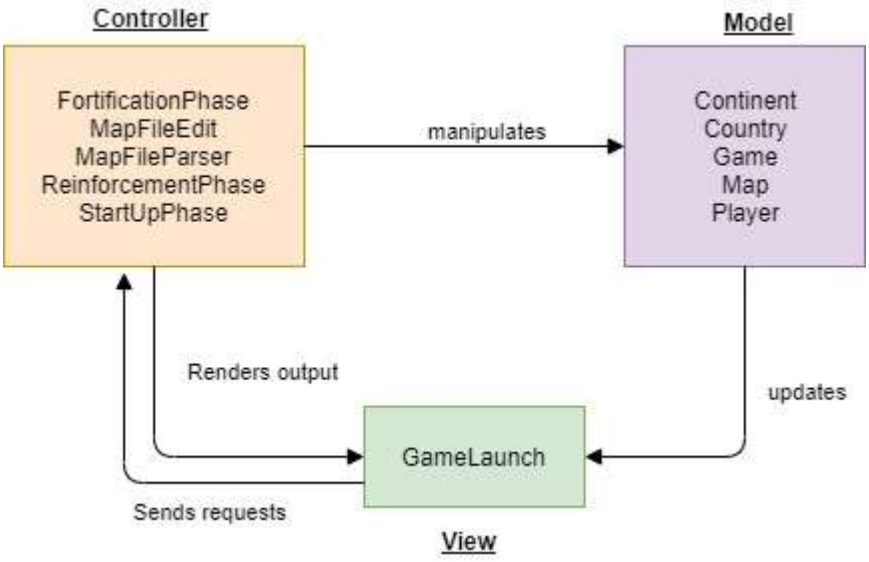


Fig. (2) Classes for Risk Game divided into MVC architecture

MODEL: Model as a whole represents and encapsulate the data specific to an application and define the logic and computation that manipulate and process that data. In this project model contains classes that contains logic for updating controller if the data changes in model.

Communication: User actions in the view layer that create or modify data are communicated through a controller object and result in the creation or updating of a model object. When a model object changes (for example, new data is received over a network connection), it notifies a controller object, which updates the appropriate view objects.

VIEW: In view, there are classes that is used to visualize data on model, a view object knows how to draw itself and can respond to user actions. A major purpose of view objects is to display data from the application's model objects and to enable the editing of that data.

Communication: View objects learn about changes in model data through the application's controller objects and communicate user-initiated changes.

CONTROLLER: In controller, there are classes that work and make changes on both model and view, this is necessary to reflect changes in model to view and also store changes that has been captured by view in model, the following classes are designed to exercise aforementioned functionality in different areas, namely Reinforcement, Fortification and attack.

Communication: A controller object interprets user actions made in view objects and communicates new or changed data to the model layer. When model objects change, a controller object communicates that new model data to the view objects so that they can display it.

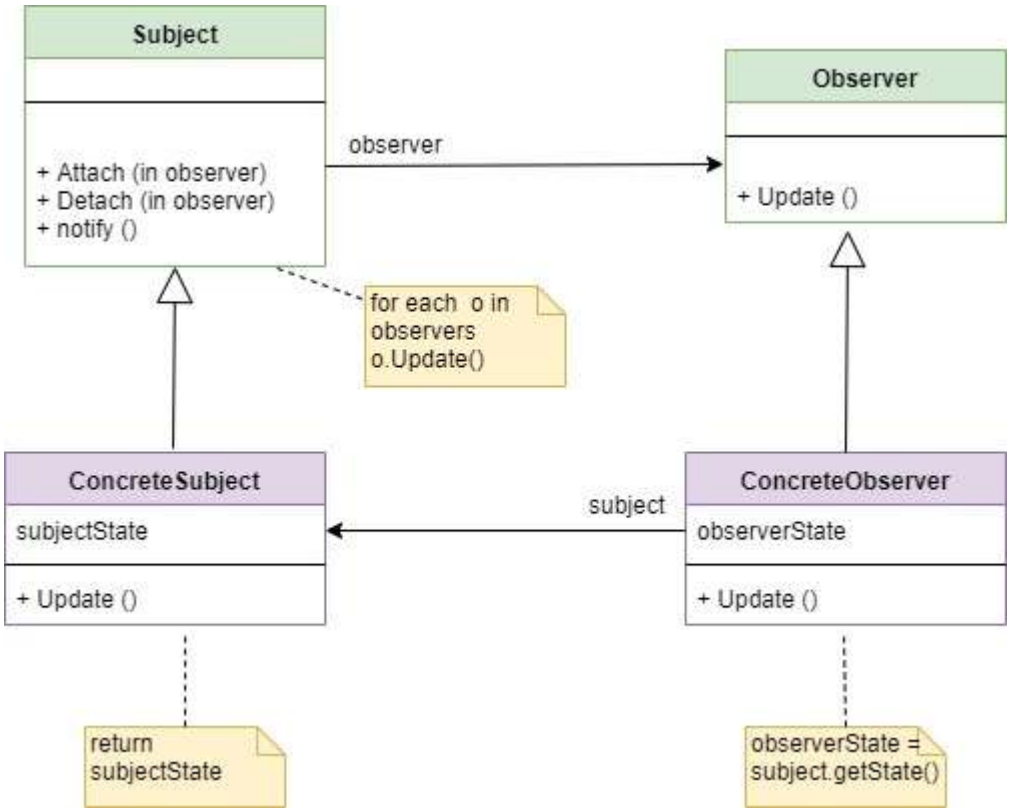
3. Observer and Observable

The Java programming language provides support for the Model/View/Controller architecture with two classes:

● Observer -- any object that wishes to be notified when the state of another object changes

● Observable -- any object whose state may be of interest, and in whom another object may register an interest.

These two classes can be used to implement much more than just the Model/View/Controller architecture. They are suitable for any system wherein objects need to be automatically notified of changes that occur in other objects.



Typically, the model is a subtype of Observable and the view is a subtype of observer. These two classes handle the automatic notification function of the Model/View/Controller architecture. They provide the mechanism by which the views can be automatically notified of changes in the model. Object references to the model in both the controller and the view allow access to data in the model.

Observer Function

public void update (Observable arg0, Object arg1)

Called when a change has occurred in the state of the observable.

How to use interface Observer and class Observable

The following section describes in detail how to create a new observable class and a new observer class, and how to tie the two together.

Extend an observable

A new class of observable objects is created by extending class Observable. Because class Observable already implements all of the methods necessary to provide the observer/observable behavior, the derived class need only provide some mechanism for adjusting and accessing the internal state of the observable object.

Below is an example of the code snippet from StartUpPhase where the observer is implemented.

```
public class StartupPhaseObserver extends PhaseViewObserver{



    public StartupPhaseObserver() {
        super();
        this.actions = new ArrayList<String>();
        this.gamePhaseName = "STARTUP PHASE";
    }

    @Override
    public void update(String action) {
        this.actions.add(action);
        System.out.println(action);

    }

    @Override
    public void setData(String playerName) {
        this.currentPlayerName = playerName;

    }

}
```

Implement an observer

A new class of objects that observe the changes in state of another object is created by implementing the Observer interface. The Observer interface requires that an *update()* method be provided in the new class. The *update()* method is called whenever the observable changes state and announces this fact by calling its *notifyObservers()* method. The observer should then interrogate the observable object to determine its new state, and, in the case of the Model/View/Controller architecture, adjust its view appropriately.

```
else if(thisInput.contains("populatecountries")) {
    this.notifyObserver( action: "Populating countries...");
    ArrayList<String> countries = new ArrayList<>();
        for(String country : Country.getListOfCountries().keySet()){
            countries.add(country);
        }
```

4. Coding Conventions

4.1 Naming Conventions

Identifier Type	Rules for Naming	Examples
Packages	Mixed case with the first letter lowercase, with the first letter of each internal word capitalized.	package com.riskGame.controller; package com.riskGame.test;
Classes	Class names should be nouns, in mixed case with the first letter of each word is capitalized. Use whole words - avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).	class GameLaunch class MapFileEdit
Interfaces	Interface names should be capitalized like class names.	interface Storing; interface Shape;

Methods	Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.	commandParser();
		editNeighbor();
		getCountryNames();

Variables	Mixed case with a lowercase first letter. Internal words start with capital letter.	int noOfPlayers; String response; int currentArmies;
Constants	The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores (“_”). (ANSI constants should be avoided, for ease of debugging.)	int MIN_WIDTH = 7; int MAX_WIDTH = 59;

4.2 Code Layout

Simple Statements

Each line must contain at most one statement only.
Example: i++; n--; // **AVOID!**
Do not used the comma operator to group multiple statements unless it is for an obvious reason.

Example: if (err) {
System.out.print(“error message :”+ error), exit(1); **//WRONG!**
}

White Spaces

i. **Blank Lines**

Blank lines improve readability by setting off sections of code that are logically related.
Two blank lines should always be used in the following circumstances:

- Between sections of a source file.
- Between class and interface definitions.

One blank line should always be used between methods.

ii. **Blank Spaces**

Blank spaces should be used in the following circumstances:
A keyword followed by a parenthesis should be separated by a space. Example:

```
while (true) {
...
}
```

Note that a blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls.

- A blank space should appear after commas in argument lists.
- All binary operators except should be separated from their operands by spaces.

Blank spaces should never separate unary operators such as unary minus, increment (“++”), and decrement (“--”) from their operands.

```
a += c + d;
a = (a + b) / (c * d);
while (d++ = s++) {
N++;
}
system.out.println("size is " + foo + "\n");
```

- The expressions in a for statement should be separated by blank spaces.

```
for (expr1; expr2; expr3)
```

- Casts should be followed by a blank. Examples:

```
myMethod((byte) aNum, (Object) x);
myFunc((int) (cp + 5), ((int) (i + 3)) + 1);
```

4.3 Comments

Comments gives overviews of code and provide additional information that is not readily available in the code itself. Comments have the information that is relevant to reading and understanding the program. For example, information about how the corresponding package is built or in what directory it resides should not be included as a comment.

Java programs can have two kinds of comments: implementation comments and documentation comments.

Documentation comments (known as "doc comments") are Java-only, and are delimited by /**...*/. Doc comments can be extracted to HTML files using the Javadoc tool.

Documentation comments are meant to describe the specification of the code, from an implementation-free perspective to be read by developers who might not necessarily have the source code at hand.

Implementation comments which are delimited by /*...*/ (for Block or Multiline comments), and // (for single line comments).

Implementation comments are meant for commenting out code or explaining about the particular implementation.

i. **Block/ Multiline Comments**
   Block comments have an asterisk "*" at the beginning of each line except the first. /* * Here is a block comment. */
   Block comments should be used at the beginning of each file and before each method. They can also be used in other places, such as within methods. Block comments inside a function or method should be indented to the same level as the code they describe. A block comment should be preceded by a blank line to set it apart from the rest of the code.
   Block comments are used to provide descriptions of files, methods, data structures and algorithms.

ii. **Single line comments**

   Short comments can appeared on a single line indented to the level of the code that follows. They are usually used to describe the working of the particular statement or explaining the details about why something is used in a certain way in the code.

4.4 Tags for Java Docs

@throws : Used to describe an exception that may be thrown from this method. Note that if you have a throws clause, Javadoc will already automatically document the exceptions listed in the throws clause.

{@inheritDoc} : Used to copy the description from an overridden method.

{@link *reference*} : Used to link to another documented symbol, or to a URL external to the documentation.

@see : This tag is used to add a hyperlink "See Also" entry to the class.

@return : Used to add a return type description for a method. This tag is meaningful only if the method's return is non-void.

Example : @return true if the array is empty; otherwise return false.

@param : This tag is used to add a parameter description for a method. This tag contains two parts: the first is the name of the parameter and the second is the description. The description can be more than one line.

Example : @param size the length of the passed array.

@author: This tag is used to create an author entry. You can have multiple **@author** tags. This tag is meaningful only for the class/interface in Java Doc comment.

@version : This tag is used to create a version entry. A Java Doc comment may contain at most one **@version** tag. Version normally refers to the version of the software (such as the JDK) that contains this feature. If you are using CVS, you can also use the following to have any CVS commit to fill in the version tag with the CVS revision number: **@version $Revision $.**

5. References

https://www.oracle.com/technetwork/java/codeconvtoc-136057.html

https://android.jlelse.eu/java-coding-standards-ee1687a82ec2

https://google.github.io/styleguide/javaguide.html

https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

https://www.codecademy.com/articles/mvc

https://www.tutorialspoint.com/design_pattern/mvc_pattern.htm

https://www.geeksforgeeks.org/loops-in-java/

https://www.javatpoint.com/java-for-loop

https://github.com/Code2Bits/Design-Patterns-Java/tree/master/Behavioral%20Patterns/Observer

https://sourcemaking.com/design_patterns/observer

https://www.geeksforgeeks.org/observer-pattern-set-1-introduction/